You may take this test with you afterwards, but you must turn in your answer sheet.

This test has the following sections:

I. True/False .................................. 80 points; (40 questions, 2 points each)
II. Multiple Choice.................. 20 points; (10 questions, 2 points each)

---------------
100 points total

This test is worth 10% of your final grade. You must put your answers on the bubble form. You are allowed to have resources with you printed on paper, but no computers. For the multiple choice problems, select the best answer for each one and select the appropriate letter on your answer sheet. Be careful - more than one answer may seem to be correct. Some questions are tricky.

**True/False: (2 points each)** On your bubble form fill out A for true and B for false.

**T F 1.** A linked-list can be the underlying structure used to implement a stack or queue.

**T F 2.** Some variant of an array can be the underlying structure used to implement a stack or queue.

**T F 3.** A profiler can identify the line or lines of code that are a bottleneck when the program is running.

**T F 4.** If for some reason you only had space to store either the circularly linked list tail pointer or the head pointer, it would be best to store the head.

**T F 5.** When converting some arbitrary infix expression to its corresponding postfix representation, a stack is the preferred data structure to use.

**T F 6.** The following shows correct dominance relationships with regards to Big-Oh:

\[ O(n!) \gg O(2^n) \gg O(n^3) \gg O(n^2) \gg O(n) \gg O(n \log n) \gg O(\log n) \gg O(1) \]

**T F 7.** A business holds a competition for an algorithm that will work with groups of no more than 1,000 items at a time. For this data your \( O(n^3) \) algorithm is faster than your friend’s is \( O(2^n) \) algorithm.

**T F 8.** When it comes to comparing programs that are in different Big-Oh dominance classes, constants can be ignored.

**T F 9.** An algorithm that runs in \( O(2n) \) runs faster than another version that runs in \( O(4n) \).

**T F 10.** An algorithm that is \( O(n \log n) \) is faster than a second algorithm which is \( O(n) \) because the value \( n \log n \) is smaller than \( n \) for a variety of numbers, such as 2,3,4,5,…

**T F 11.** \( n \) unique values could be sorted in \( O(n) \) time by using an additional array of size \( n^2 \)
T F 12. Quicksort is $O(n \log n)$.

T F 13. The complexity of an algorithm can be greatly reduced if we change the type of data structure used to represent the algorithm.

T F 14. Given a proper implementation of a stack, along with properly defined operators $push(\ldots)$ and $pop(\ldots)$, the code shown below would allow us to use a manually implemented stack to reverse input, so that input of: $abc\$ would give as output: $cba$

```c
void reverseInputUsingExplicitStack()
{
    Node *pTop = NULL;
    char c;

    do {
        cin >> c;
        if( c!='$') {
            push( pTop, c);
        }
    } while (c!='$');

    while( pTop!=NULL) {
        cout << pop(pTop);
        pTop=pTop->pNext;
    }
}
```

T F 15. A node (vertex) can have a name and carry other associated information.

T F 16. An edge is a connection between two vertices.

T F 17. A path is a list of distinct vertices in which successive vertices are connected by edges.

T F 18. A tree is a set of nodes connected by vertices where there is precisely one path connecting any two nodes.

T F 19. Parent nodes are always above their children nodes.

T F 20. All nodes with no children are leaf nodes, regardless of their level in the tree.

T F 21. The smallest node in a tree will always be to the left of the root node.

T F 22. A binary tree is a tree where each node has at most two children.

T F 23. A pre-order binary tree traversal could be written without using recursion or any additional data structure (e.g. stack, array, linked list…) as long as you implement a parent pointer for each node.

T F 24. Given an unsorted binary tree of $n$ randomly distributed integer values, it is possible to use only the $inOrderTraversal(\ldots)$ and the $insert(\ldots)$ functions of a binary search tree to print out those $n$ values in sorted order in $O(n \log n)$. 
T F 25. Given an unsorted binary tree of \( n \) randomly distributed integer values, it is possible to use only the \textit{minimum}(...) , \textit{successor}(...) and the \textit{insert}(...) functions of a binary search tree to print out those \( n \) values in sorted order in \( O(n \log n) \).

T F 26. Given an unsorted binary tree of \( n \) randomly distributed integer values, it is possible to use only the \textit{minimum}(...) , \textit{delete}(...) and the \textit{insert}(...) functions of a binary search tree to print out those \( n \) values in sorted order in \( O(n \log n) \).

T F 27. Every set of values will give the identical binary search tree, regardless of the order in which the values are presented when building the tree.

T F 28. Assume you have a binary search tree with \( O(\log n) \) for search, where the Node declaration contains a field for \textit{data}, and a field each for \textit{leftChild} and \textit{rightChild}, there is a tree root pointer and there are no other pointers into parts of the tree. Assuming some node \( A \) is already found, implementing \textit{Node * predecessor(A)} in \( O(1) \) requires modifying the Node declaration.

T F 29. Assume a binary search tree has been built using \( n \) integer values. Regardless of the distribution of the input values used to build the tree, those values can be printed out in order in \( O( n) \) time.

T F 30. Assume a binary search tree has been built using \( n \) integer values. Regardless of the distribution of the input values used to build the tree, any value can be found in \( O( n) \) time.

T F 31. Assume a binary search tree has been built using \( n \) integer values. Regardless of the distribution of the input values used to build the tree, any new value can be added in \( O( \log n) \) time.

T F 32. Assume you have a stream of \( n \) integer values that you want to insert into a binary search tree. Each successive value alternates between being the smallest and the largest value seen so far. Each new value can be added in \( O( \log n) \) time.

T F 33. Assuming you are writing a program where a very large number of values are read in at the beginning of the day where time is not at all an issue. Then throughout the day the smallest value is displayed and then deleted, where doing this as quickly as possible is extremely important. Implementing this as a linked list of sorted values will give the same performance as implementing it using a binary search tree.

T F 34. Nested calls to function \textit{newNode}(...) shown below could be used to build a tree.

```cpp
Node * newNode( int value, Node *left, Node *right)
{
    Node *pTemp = new Node;
    pTemp->data = value;
    pTemp->pLeft = left;
    pTemp->pRight = right;

    return pTemp;
}
```

T F 35. In a binary search tree, for any node \( pTemp \) we can find the successor of \( pTemp \) using:

\[
\text{getMin( } pTemp->pRight \text{)}
\]
T F 36. The following code to do an in-order traversal of a binary search tree could be rewritten without using recursion:

```c
void inOrderTraversal( Node * pRoot) {
   if( pRoot != NULL) {
      inOrderTraversal( pRoot->pLeft); // recurse down to the left
      cout << pRoot->data << " "; // display contents
      inOrderTraversal( pRoot->pRight); // recurse down to the right
   }
} // end inOrderTraversal(...)
```

T F 37. The following code works properly to search for a value in a binary search tree:

```c
Node * searchTree( Node *pRoot, int searchValue) {
   if( pRoot->data == searchValue) {
      return pRoot;
   } else if( searchValue < pRoot->data) {
      return searchTree( pRoot->pLeft, searchValue);
   } else {
      return searchTree( pRoot->pRight, searchValue);
   }
   return NULL;
}
```

T F 38. The following code works properly to allow inserting duplicate values into a binary search tree:

```c
void insertIntoTree( Node *&pRoot, int value) {
   if( pRoot == NULL) {
      pRoot = newNode( value, NULL, NULL);
   } else if( value < pRoot->data) {
      insertIntoTree( pRoot->pLeft, value);
   } else {
      insertIntoTree( pRoot->pRight, value);
   }
}
```
T F 39. The following code works properly to allow inserting values into a binary search tree, disallowing duplicates:

```cpp
Node * insertIntoTree2( Node *pRoot, int value)
{
    if( pRoot == NULL) {
        pRoot = newNode( value, NULL, NULL);
    } else if( value == pRoot->data) {
        cout << "No insert, value is already in tree.\n";
    } else if( value < pRoot->data) {
        pRoot->pLeft = insertIntoTree2( pRoot->pLeft, value);
    } else {
        pRoot->pRight = insertIntoTree2( pRoot->pRight, value);
    }
}
```

T F 40. The following code works properly to find the parent node of some given node pTemp

```cpp
void findParent2( Node *pRoot, Node *pTemp, Node *&pParent)
{
    if( pRoot == NULL) {
        return;
    } else if( pRoot->pLeft == pTemp || pRoot->pRight == pTemp) {
        pParent = pRoot;
    } else {
        findParent2( pRoot->pLeft, pTemp, pParent);
        if( pParent == NULL) {
            findParent2( pRoot->pRight, pTemp, pParent);
        }
    }
}
```

Multiple Choice (2 points each)

41) The expression \( A \times B + C \) converted to postfix is:

a) \( ABC^+ \)
b) \( AB^*C+ \)
c) \(^*AB+C\)
d) \(^+ABC\)
e) None of the above

42) The expression \( (A + B) \times D + E / (F + A \times D) + C \) converted to postfix is:

a) \( AB+D*E+FAD^+*/C+ \)
b) \( AB^*+EFA^+*/+C+ \)
c) \( ABD^*+EFAD^+*/C+ \)
d) \( A+BDA^+*/FAD^+*/C+ \)
e) None of the above
43) Consider the following code used to convert a properly formatted infix expression to postfix:

```c
void convertInfixToPostfix( char inputLine[])
{
    Node *pTop = NULL;     // create the stack top pointer
    char c;              // current input character
    for(int i=0; i<strlen(inputLine); i++) {
        // get next input character
        c = inputLine[i];
        // Handle operators
        else if(c == '+' || c == '-' || c == '*' || c == '/')
        {
            while( pTop!=NULL &&
                (operatorWeight(pTop->data) >= operatorWeight(c)) )
            {
                cout << pop( pTop);
            }
            push( pTop, c);
        }
        else if( c>='0' && c<='9')
        {
            cout << c;
        }
    //end for( int i...
    while(pTop != NULL) {
        cout << pop( pTop);
    }
}
```

For the sample input: \(2*3*4+5\) what is the effect if the \(\geq\) in the inner while loop (pointed to by the arrow) is converted to \(>\), assuming the rest of the code is correct, as seen in class?

a) Multiplication and addition have the same order of precedence.
b) Addition gets precedence over multiplication
c) Precedence within multiplication ends up right-to-left instead of left-to-right
d) All operators have the same order of precedence
e) The output is not in postfix form

44) What is the order of complexity (Big-Oh) for the following section of code?

```c
for( int i=0; i<n; i++) {
    for( int j=0; j<i; j++) {
        cout << i*j << " ";
    }
    cout << endl;
}
```

a) \(O(c)\)
b) \(O(n^2)\)
c) \(O(n \log n)\)
d) \(O(\log n)\)
e) \(O(n)\)
Consider the following code:

```c
void sort1( int theArray[], int arraySize)
{
    for( int pass=1; pass < arraySize; pass++) {
        for( int current=0; current < arraySize-pass; current++) {
            if ( theArray[ current] > theArray[ current+1]) {
                swap( theArray, current, current+1);
            }
        }
    }
}
```

The run-time complexity (Big-Oh) for the above code is:

a) Constant: O(1)  
b) Linear: O(n)  
c) Logarithmic: O(log n)  
d) Loglinear: O(n log n)  
e) Quadratic: O(n^2)

Consider the following code that is passed an array bigger than 3 elements:

```c
void sample( int theArray[], int arraySize)
{
    for( int pass=1; pass < arraySize; pass++) {
        for( int current=0; current < 1; current++) {
            if ( theArray[ current] > theArray[ current+1]) {
                swap( theArray, current, current+1);
            }
        }
    }
}
```

The run-time complexity (Big-Oh) for the above code is:

a) Constant: O(1)  
b) Linear: O(n)  
c) Logarithmic: O(log n)  
d) Loglinear: O(n log n)  
e) Quadratic: O(n^2)

Assume you have a stream of n integer values in ascending order that you want to insert into a binary search tree. What is the run-time complexity (Big-Oh) of running this program on this data?

a) O( c)  
b) O(n^2)  
c) O(n log n)  
d) O( log n)  
e) O(n)
48) Assume you have a stream of \( n \) integer values in random order that you want to insert into a binary search tree. What is the run-time complexity (Big-Oh) of running this program on this data?

   a) \( O( c) \)
   b) \( O( n^2) \)
   c) \( O( n \log n) \)
   d) \( O( \log n) \)
   e) \( O( n) \)

49) Consider the following code:

```c
void sort2( int theArray[], int arraySize)
{
    int indexOfSmallest = 0;
    
    for( int pass=0; pass < arraySize; pass++ ) {
        indexOfSmallest = pass;
        for ( int i=pass+1; i < arraySize; i++) {
            if( theArray[ i] < theArray[ indexOfSmallest] ) {
                indexOfSmallest = i;
            }
        }
        swap( theArray, pass, indexOfSmallest);
    }
}
```

The run-time complexity (Big-Oh) for the above code is:

   a) Constant: \( O(1) \)
   b) Linear: \( O( n) \)
   c) Logarithmic: \( O( \log n) \)
   d) Loglinear: \( O( n \log n) \)
   e) Quadratic: \( O( n^2) \)

50) Consider a binary tree \( T \) in which all the levels are completely full and the nodes are numbered in a breadth-first fashion, where the root is 1, its immediate left and right children are 2 and 3, and their children respectively are 4, 5 and 6, 7, and so on. Which of the following are true about \( T \)?

   A. It could be implemented using a static (not dynamic) array, but the total number of Nodes must be known.
   B. Implementation using an array would require less storage than using a tree of Node structures.
   C. In an array implementation, which level a node is in can be determined by its node number.
   D. Accessing any node can be done in constant time \( O( c) \).

   a) A, B
   b) A, C, D
   c) A, B, D
   d) B, C, D
   e) A, B, C, D